1. **Explain about Thread Synchronisation?**

Thread synchronization is a mechanism to ensure that multiple threads do not interfere with each other while accessing shared resources. It prevents race conditions, maintains data consistency, and allows proper execution order.

Example: Synchronizing a shared counter
Consider a scenario where multiple threads increment a shared counter. Without synchronization, inconsistent results may occur due to race conditions.

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;  // Only one thread can execute this at a time
    }

    public synchronized int getCount() {
        return count;
    }
}

class Worker extends Thread {
    private Counter counter;

    public Worker(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}

public class ThreadSyncExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Worker(counter);
        Thread t2 = new Worker(counter);

        t1.start();
        t2.start();

        t1.join();
```

```
        t2.join();

        System.out.println("Final counter value: " + counter.getCount());
    }
}
```

Explanation:

- ✓ synchronized ensures that only one thread modifies count at a time.
- ✓ join() ensures that both threads complete execution before printing the final count.
- ✓ Without synchronization, unpredictable values may appear due to concurrent access.

2. Explain Synchronization block?

Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks. You keep shared resources within this block.

```
class SharedResource {
    private int count = 0;

    public void increment() {
        synchronized (this) {  // Lock is only on this block
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}
```

3. Explain Synchronised Method?

In Java, method synchronization is achieved by adding the **synchronized k**eyword. Then the entire method is treated as a critical section. Ensuring that only one thread can execute that method at any given time. This approach locks the entire method, preventing other threads from accessing it until the first thread finishes its execution.

```
class SharedResource {
    public synchronized void displayMessage(String message) {
        System.out.println("[" + message);
        try {
            Thread.sleep(1000); // Simulate some work
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("]");
    }
}
```

4. Explain Class level synchronisation?

Synchronization can be applied at the class level, meaning all threads accessing the class share the same lock.

```
public class StaticVariableLock implements Runnable {
        private static Object staticVariable=new Object();
        public static void main(String[] args) {
                Thread t1 = new Thread(new StaticVariableLock(),"Thread T1");
                Thread t2 = new Thread(new StaticVariableLock(),"Thread T2");
                t1.start();
                t2.start();
        }
        private static void staticVariableMethod() {
                synchronized (StaticVariableLock.staticVariable) { //Applied lock on static variable
                        try {
                                System.out.println(Thread.currentThread().getName() +": entered");
                                Thread.sleep(1000);// Goes for sleep of 1s
                                System.out.println(Thread.currentThread().getName() +": exited");
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
        @Override
        public void run() {
                StaticVariableLock.staticVariableMethod();
        }
}
```

Output:

```
Thread T1: entered
Thread T1: exited
Thread T2: entered
Thread T2: exited
```

5. Is deadlock occurs during thread synchronisation?

Yes, deadlocks can occur during thread synchronization when multiple threads hold locks on shared resources and wait indefinitely for each other to release their locks. This results in a situation where no thread can proceed, leading to a system freeze or performance degradation.

```
class Resource {
    public void methodA(Resource other) {
        synchronized (this) {
            System.out.println(Thread.currentThread().getName() + " locked " + this);
            try { Thread.sleep(100); } catch (InterruptedException ignored) {}

            synchronized (other) {  // Waiting for other resource
                System.out.println(Thread.currentThread().getName() + " locked " + other);
            }
        }
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        Resource r1 = new Resource();
        Resource r2 = new Resource();

        Thread t1 = new Thread(() -> r1.methodA(r2), "Thread-1");
        Thread t2 = new Thread(() -> r2.methodA(r1), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

How Deadlock Happens:

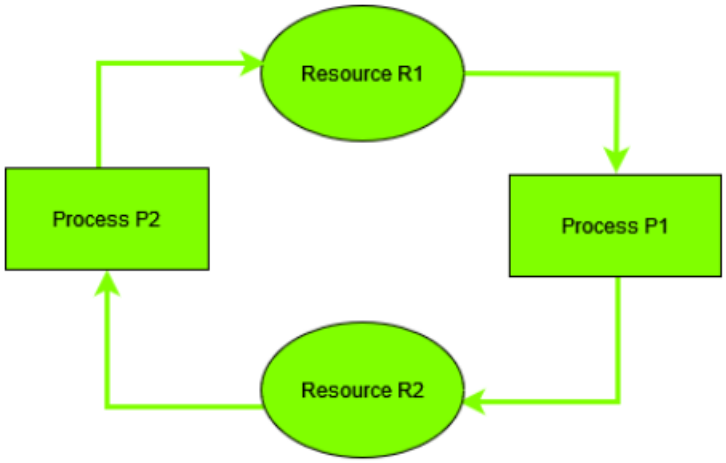
1. `Thread-1` locks `r1` and waits for `r2`.
2. `Thread-2` locks `r2` and waits for `r1`.
3. Since neither can proceed, they remain blocked forever.

6. How to avoid deadlock in thread synchronisation?

✅ **Avoid Nested Locks:** Minimize locking multiple resources simultaneously.
✅ **Lock Ordering:** Always acquire locks in a consistent order.
✅ **Use TryLock (ReentrantLock):** Allows timeout to break potential deadlock.
✅ **Deadlock Detection Tools:** Use monitoring tools to identify deadlocks in production.

7. Explain about Live Lock ?

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.



Process P1 holds resource R1 & requesting for R2
Process P2 holds resource R2 & requesting for R1

Example 1:

An easiest example of Livelock would be two people who meet face-to-face in a corridor, and both of them move aside to let the other pass. They end up moving from side to side without making any progress as they move the same way at the time. Here, they never cross each other.

8. What is Starvation?

Starvation is a situation where all the low priority processes got blocked, and the high priority processes proceed. In any system, requests for high/low priority resources keep on happening dynamically. Thereby, some policy is require to decide who gets support when.

Why Starvation Happens in This Example

1. The `High-Priority-Thread` keeps acquiring the lock.
2. The `Low-Priority-Threads` rarely get a chance to run.
3. Eventually, the low-priority threads may **starve** indefinitely.

How to Prevent Starvation

✅ **Use Fair Locks (** `ReentrantLock(true)` **)** – Ensures fairness in thread execution.
✅ **Avoid Priority-Based Scheduling** – Favor balanced scheduling policies.
✅ **Introduce Randomized Delays** – Helps lower-priority threads get execution time.
✅ **Use Aging Mechanism** – Gradually increases priority of waiting threads to prevent starvation.

9. Explain Semaphore in threads?

A **Semaphore** is a synchronization mechanism that controls access to a shared resource by multiple threads using **permits**. It helps **limit concurrent access** and prevents race conditions.

```
import java.util.concurrent.Semaphore;

class SharedResource {
    private final Semaphore semaphore = new Semaphore(2); // Only 2 permits

    public void accessResource() {
        try {
            semaphore.acquire(); // Request a permit
            System.out.println(Thread.currentThread().getName() + " is accessing the resource.");
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException ignored) {
        } finally {
            System.out.println(Thread.currentThread().getName() + " is releasing the resource.");
            semaphore.release(); // Release permit
        }
    }
}

public class SemaphoreExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        for (int i = 1; i <= 5; i++) {
            new Thread(resource::accessResource, "Thread-" + i).start();
        }
    }
}
```

How It Works

1. `Semaphore(2)` allows only **two** threads to enter at a time.
2. If more threads try to access the resource, they **wait until a permit is available**.
3. Once a thread **releases a permit**, another thread can acquire it.

10. Difference between Semaphore and synchronised in threads?

- **synchronized** – Method can be accessed by one thread at a time
- **Semaphore** – Method can be accessed by n threads at a time. (n specified while initializing Semaphore)

11. Explain Atomic Variables in threads?

Regular variables like int or long can be modified by multiple threads simultaneously, causing inconsistencies. Atomic variables ensure **thread-safe** updates without using synchronized.

Java provides the java.util.concurrent.atomic package, which offers **atomic variables** that help prevent race conditions **without using explicit synchronization**.

1. **set(int value):** Sets to the given value
2. **get():** Gets the current value
3. **lazySet(int value):** Eventually sets to the given value
4. **compareAndSet(int expect, int update):** Atomically sets the value to the given updated value if the current value == the expected value
5. **addAndGet(int delta):** Atomically adds the given value to the current value
6. **decrementAndGet():** Atomically decrements by one the current value

```java
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet(); // Atomic operation, no need for synchronized
    }

    public int getCount() {
        return count.get();
    }
}

public class AtomicExample {
    public static void main(String[] args) throws InterruptedException {
        AtomicCounter counter = new AtomicCounter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

12. Explain about Executor framework and its example?

The Java Executor framework, introduced in JDK 5, is a powerful tool for managing and executing tasks asynchronously. It is used to run the Runnable objects without creating new threads every time and mostly reusing the already created threads.

Types of Executors in Java

Below are the commonly used executor types:

1. SingleThreadExecutor
2. FixedThreadPool(n)+
3. CachedThreadPool
4. ScheduledExecutor

Let us discuss these popular Java executors in some detail, and what exactly they do to get a better idea before implementing the same.

1. SingleThreadExecutor

A thread pool of a single thread can be obtained by calling the static *newSingleThreadExecutor()* method of the Executors class. It is used to execute tasks sequentially.

Syntax:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

2. FixedThreadPool

As the name indicates, it is a thread pool of a fixed number of threads. The tasks submitted to the executor are executed by the n threads, and if there are more task, they are stored on a **LinkedBlockingQueue**. It uses Blocking Queue.

Syntax:

```
ExecutorService fixedPool = Executors.newFixedThreadPool(2);
```

3. CachedThreadPool

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. It uses a SynchronousQueue queue.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

4. ScheduledExecutor

Scheduled executors are based on the interface ScheduledExecutorService which extends the ExecutorService interface. This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.

```
ScheduledExecutorService scheduledExecService = Executors.newScheduledThreadPool(1);
```

The tasks can be scheduled using either of the two methods:

- **scheduleAtFixedRate**: This schedules tasks to start at fixed intervals, but if a task takes longer than the interval, the next execution will be delayed. It does not interrupt the running task, but, upcoming executions are queued and wait for the current one to finish
- **scheduleWithFixedDelay**: This will start the delay countdown only after the current task completes.

Syntax:

```
scheduledExecService.scheduleAtFixedRate
(Runnable command, long initialDelay, long period, TimeUnit unit)
```

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class WorkerTask implements Runnable {
    private final int taskId;

    public WorkerTask(int taskId) {
        this.taskId = taskId;
    }

    @Override
    public void run() {
        System.out.println("Task " + taskId + " executed by " + Thread.currentThread().getName());
    }
}

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3); // Pool with 3 threads

        for (int i = 1; i <= 5; i++) {
            executor.submit(new WorkerTask(i)); // Submitting tasks
        }

        executor.shutdown(); // Gracefully shutting down the executor
    }
}
```

How This Works

1. **Creates a thread pool** of **3 worker threads**.
2. **Submits 5 tasks**, distributing them across available threads.
3. **Manages automatic thread reuse**, improving performance.
4. **Shuts down gracefully** using `shutdown()` after all tasks complete.

13. Difference between runnable and callable?

Feature	Runnable	Callable<T>
Return Type	Does **not** return a result (void)	Returns a result of type `T`
Exception Handling	Cannot throw checked exceptions	Can throw checked exceptions
Thread Execution	Used with `Thread` or `ExecutorService.execute()`	Used with `ExecutorService.submit()`
Method Definition	Defines `run()` method	Defines `call()` method
Usage	Ideal for simple tasks that do not require a result	Suitable for tasks that return results or handle exceptions

Example Runnable:

```
class MyTask implements Runnable {
    @Override
    public void run() {
        System.out.println("Task executed by " + Thread.currentThread().getName());
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyTask());
        thread.start();
    }
}
```

✅ **Does not return a result. Simply executes a task.**

Example Callable:

```
import java.util.concurrent.*;

class MyCallableTask implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        return 10 * 2; // Returns computed result
    }
}

public class CallableExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Integer> future = executor.submit(new MyCallableTask());

        System.out.println("Result: " + future.get()); // Retrieves result

        executor.shutdown();
    }
}
```

✅ **Returns a value and handles exceptions. Uses Future to get the result asynchronously.**

14. How to resolve producer and consumer problem in the threads?

The **Producer-Consumer Problem** is a classic synchronization issue in multithreading where:

- The **Producer** creates data and puts it into a shared resource (like a queue).
- The **Consumer** takes data from the shared resource.
- Proper synchronization is needed to ensure that:
  - The producer doesn't add data when the queue is full.
  - The consumer doesn't remove data when the queue is empty.

2. Using `BlockingQueue` (Better Approach)

The `BlockingQueue` handles synchronization internally and prevents race conditions **without using** `wait()` **and** `notify()`.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

class ProducerConsumerQueue {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    queue.put(i); // Blocks if the queue is full
                    System.out.println("Produced: " + i);
                    Thread.sleep(500);
                }
            } catch (InterruptedException ignored) {}
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 10; i++) {
                    int item = queue.take(); // Blocks if the queue is empty
                    System.out.println("Consumed: " + item);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException ignored) {}
        });

        producer.start();
        consumer.start();
    }
}
```

15. Difference between thread local and inherited thread local?

Both ThreadLocal and InheritedThreadLocal provide thread-local variables that help maintain separate copies for each thread. However, InheritedThreadLocal extends the concept by allowing **child threads to inherit values** from their parent threads.

Feature	`ThreadLocal<T>`	`InheritedThreadLocal<T>`
Purpose	Provides a variable local to a thread	Allows child threads to inherit values from parent thread
Value Inheritance	Values are **not** passed to child threads	Child threads **inherit** values from parent thread
Usage Scenario	Use when each thread requires isolated data	Use when child threads need initial data from parent threads
Example Use Case	Per-thread database connection, user sessions	Transaction propagation, request tracking in multithreading

Choosing Between `ThreadLocal` and `InheritedThreadLocal`

- **Use** `ThreadLocal` when threads need completely **independent copies**.
- **Use** `InheritedThreadLocal` when child threads **must inherit values** from parent threads.

16. Explain about virtual threads?

In Java, Virtual threads are now supported by the Java Platform. Virtual threads are lightweight threads that greatly minimize the effort required to create, operate, and manage high volumes systems that are concurrent. As a result, they are more efficient and scalable than standard platform threads.
A virtual thread is an instance of java.lang.Thread, independent of any OS thread, is used to run programs. The Java runtime suspends the virtual thread until it resumes when the code calls a blocked I/O operation. Virtual threads have a limited call stack and can only execute one HTTP client call or JDBC query. They are suitable for delayed operations, but not for extended CPU-intensive tasks.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class VirtualThreadExample {
    public static void main(String[] args) {
        try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
            for (int i = 1; i <= 5; i++) {
                executor.submit(() -> System.out.println("Task running on " + Thread.currentThread()));
            }
        }
    }
}
```

Why Use Virtual Thread?

In your high quantities concurrent applications, use virtual threads if you have many concurrent processes that take a long time to finish. Server applications often handle large numbers of client requests, which requires blocking I/O tasks such as resource access. This makes server applications high-throughput.

Virtual threads do not execute code more quickly than platform threads. Their goal is to provide scale (greater throughput) rather than speed (lower latency).

17. Difference between synchronised, volatile and Atomic?

Features	Synchronized	Volatile	Atomic
Applies to	It applies to only blocks or methods.	It applies to variables only.	It is also applicable to variables only.
Purpose	It ensures mutual exclusion and data consistency by acquiring locks.	It ensures the visibility of variables across threads but does not guarantee atomicity	It provides atomic operations on variables without needing locks
Performance	Performance is relatively low compared to volatile and atomic keywords because of the acquisition and release of the lock.	Performance is relatively high compared to synchronized Keyword.	Performance is relatively high compared to both volatile and synchronized keyword.
Concurrency	Because of its locking nature, it is not immune to concurrency hazards such as deadlock and livelock.	Because of its non-locking nature, it is immune to concurrency hazards such as deadlock and livelock.	Because of its non-locking nature, it is immune to concurrency hazards such as deadlock and livelock.